MIPS Functions and Instruction Formats

The Contract:
The MIPS Calling Convention

- You write functions, your compiler writes functions, other compilers write functions…
  - And all your functions call other functions which call other functions
- We want them all to play nicely together
- Thus the MIPS Calling Convention
  - How do you pass arguments?
  - What registers and state ***are not changed*** between when you call the function and when it returns?

Why We Need it?


- ```
  int sumSquare(int x, int y) {
    return mult(x,x)+ y;
  }
  ```
  - What happens when a function calls another function?
    - Would clobber values in $a0 to $a3 and $ra
- What is the solution?
  - Need to ***save values on the stack***
  ```

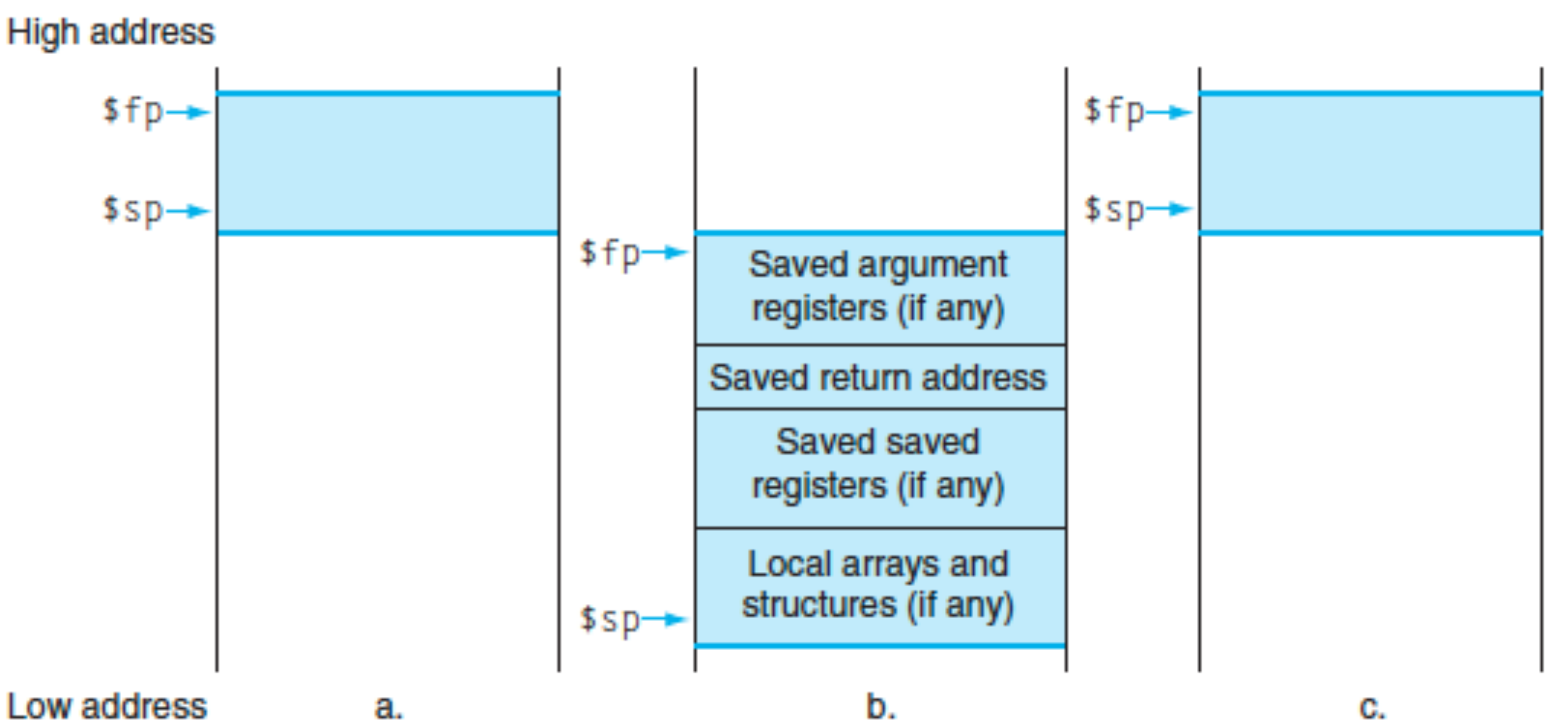
Optimized Function Convention

- To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories
- Preserved across function call (Callee-saved)
  - Calling function can rely on values being unchanged when the called function returns
  - **`$sp`**, **`$gp`**, **`$fp`**, "saved registers" **`$s0-$s7`**
- Not preserved across function call (Caller-saved)
  - Caller cannot rely on values being unchanged, so the caller must save them if needed across a function call
  - Return value registers **`$v0,$v1`**, Argument registers **`$a0-$a3`**, **`$t0-$t9`**,**`$ra`**

Allocating Space on Stack

- C has two storage classes: automatic and static
  - ***Automatic*** variables are local to function and discarded when function exits
  - Static variables exist across exits from and entries to procedures
- Use the stack for automatic (local) variables that don't fit in registers
- Use the stack for all callee-saved registers used in the function
  - And then restore those values upon function exit
- Procedure frame or activation record: segment of stack with saved registers and local variables
- Some MIPS compilers use a frame pointer (**`$fp`**) to point to first word of frame
  - But in general we don't since, at any given point in the code, there is a fixed difference between the frame pointer and the stack pointer (**`$sp`**)

Stack Before, During, After Call



Using the Stack (1/2)

- So we have a register **`$sp`** which always points to the last used space in the stack.
- To use the stack, we decrement this pointer by the amount of space we need and then fill it with info
  - And then when done, we restore anything that needs restoring before exit
- So, how do we compile this?
  - ```
    int sumSquare(int x, int y) {
        return mult(x,x)+ y;
    }
    ```
    ```

Using the Stack (2/2)

```
int sumSquare(int x, int y) {
  return mult(x,x)+ y; }
```

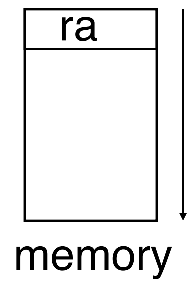

- Hand-compile

```
sumSquare:
         addi $sp,$sp,-8 # space on stack
"push"   sw $ra, 4($sp)   # save ret addr
         sw $a1, 0($sp)   # save y
         add $a1,$a0,$zero # mult(x,x)
         jal mult         # call mult
         lw $a1, 0($sp)   # restore y
         add $v0,$v0,$a1 # mult()+y
         lw $ra, 4($sp)   # get ret addr
"pop"    addi $sp,$sp,8  # restore stack
         jr $ra
mult:    ...
```

Basic Structure of a Function

Prologue

```
entry_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)   # save $ra
save other regs if need be
```



...

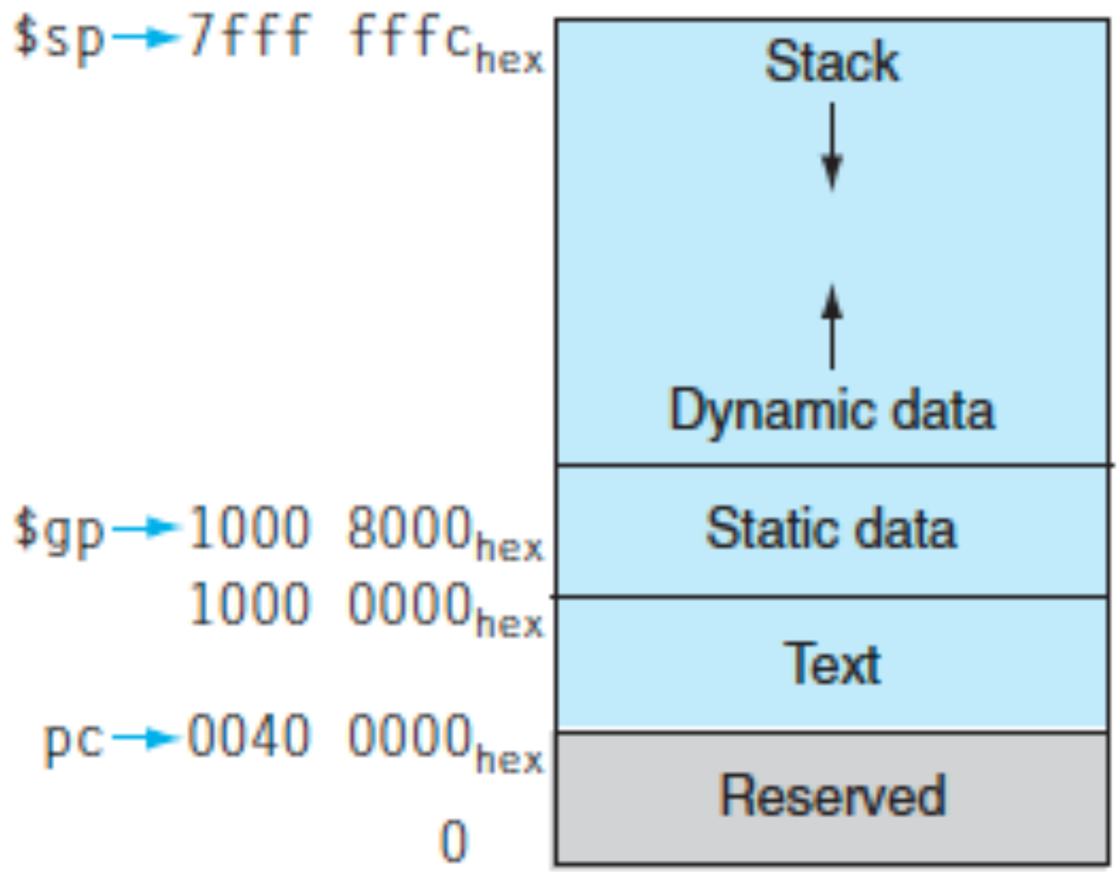
Body **(call other functions…)**

Epilogue

```
restore other regs if need be
lw $ra, framesize-4($sp)   # restore $ra
addi $sp,$sp, framesize
jr $ra
```

MIPS Default Memory Allocation

- Text segment contains the code
  - PC points to the start of it initially
- Stack grows down
  - `$sp` points to the lowest element
- `$gp` points to the start of static data



Register Allocation and Numbering

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

- $1 reserved for assembler (**`$at`**), $26 and $27 for the interrupt handler (**`$k0`** and **`$k1`**)

Recursive Function Factorial


```
int fact (int n)
{
  if (n < 1) return (1);
       else return (n * fact(n-1));
}
```

Recursive Function Factorial


```
Fact:
   # adjust stack for 2 items
   addi $sp,$sp,-8
   # save return address
   sw $ra, 4($sp)
   # save argument n
   sw $a0, 0($sp)
   # test for n < 1
   slti $t0,$a0,1
   # if n >= 1, go to L1
   beq $t0,$zero,L1
   # Then part (n==1) return 1
   addi $v0,$zero,1
   # pop 2 items off stack
   addi $sp,$sp,8
   # return to caller
   jr $ra
```

```
L1:
  # Else part (n >= 1)
  # arg. gets (n – 1)
  addi $a0,$a0,-1
  # call fact with (n – 1)
  jal Fact
  # return from jal: restore n
  lw $a0, 0($sp)
  # restore return address
  lw $ra, 4($sp)
  # adjust sp to pop 2 items
  addi $sp, $sp,8
  # return n * fact (n – 1)
  mul $v0,$a0,$v0
  # return to the caller
  jr $ra
```

mul is a pseudo instruction

The “Stack Overflow” Attack

- Recall that C allocates variables on the stack
  - And many c functions don’t actually check that they are writing into valid memory…
- So what happens if you allocate an array on the stack…
  - And then call something that writes beyond the stack…

```
void foo(){
    char bar[32];
    …
    gets(bar);
    …
}
```

The Stack Overflow Continued…


```
foo:addi $sp $sp -36
    # allocate space for
    # $ra and bar
    sw $ra $sp(32)
    …
    # bar == $sp + 0
    addi $a0 $sp $0
    jal gets
    …
    lw $ra $sp(32)
    add $sp $sp 36
    jr $ra
```

…
…
`$ra`
`bar[28:31]`
`bar[24:27]`
`bar[20:23]`
`bar[16:19]`
`bar[12:15]`
`bar[8:11]`
`bar[4:7]`
`bar[0:3]`
…

- Now what if a "bad dude" choses the input into **`gets`**?
  - Well, they can overwrite $ra and also add their own code past that point…
  - And make $ra point to their own code…
- Voila, they've taken control!

Oh, and jalr...

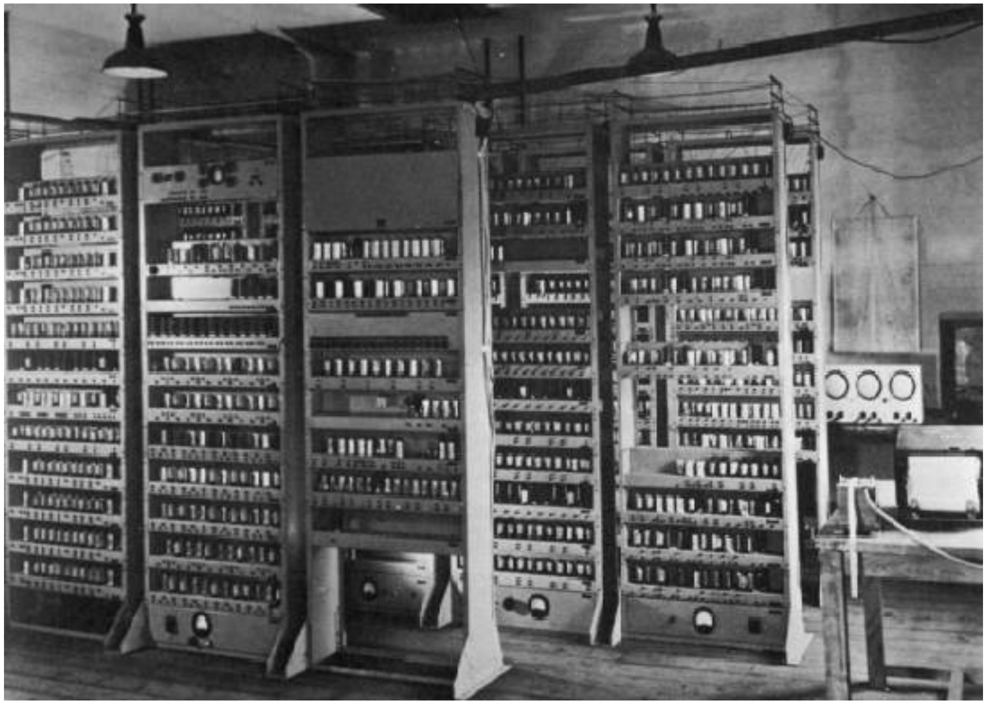
- We have `j`
  - "Jump to fixed address"
- **`jr`**
  - "Jump to address specified in register"
- **`jal`**
  - "Jump to this location and store PC+4 in $ra"
- But we need one more: Jump and Link Register, **`jalr`**
  - "Jump to this register location and store PC+4 in $ra"
- This is how we implement the pointers-to-functions ninjutsu!
  - ```
    char (*f)(char *, char *) = &foo
    ....
    (*f)("arg1", "arg2") -> jalr $whateverFisStoredAt
    ```
    ```

Clickers/Peer Instruction

- Which Statement is True?
  - a: **`$sp`** points to the lowest address currently in use on the stack
  - b: **`$ra`** stores PC+4 saved by **`jal`** so that a program knows where to return to when a function exits
  - c: **`$t0-$t9`** are callee saved registers
  - d: The classic stack overflow attack overwrites the saved return address on the stack with a new location
  - e: Nick likes trolling students on clicker questions

EDSAC (Cambridge, 1949)
First General Stored-Program Computer

- Programs held as numbers in memory
- 35-bit binary 2's complement words



Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
  - both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limited in Java by language design
- One register keeps address of instruction being executed: "Program Counter" (PC)
  - Basically a pointer to memory: Intel calls it Instruction Pointer (a better name)

Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
    - The "ABI": Application Binary Interface is a function of ***both*** the instruction set and the underlying operating system
  - Different binaries for Macintoshes and PCs
  - Different binaries for Linux i86 and Linux ARM
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
  - Leads to "backward-compatible" instruction set evolving over time
- Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set;
  the hardware can still run programs from a 1981 PC today

Instructions as Numbers (1/2)

- Currently all data we work with is in words (32-bit chunks):
  - Each register is a word.
  - **`lw`** and **`sw`** both access memory one word at a time.
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so "**`add $t0,$0,$0`**" is meaningless.
  - MIPS/RISC seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also

Instructions as Numbers (2/2)

- One word is 32 bits, so divide instruction word into “fields”.
- Each field tells processor something about instruction.
- We could define different fields for each instruction, but MIPS seeks simplicity, so define 3 basic types of instruction formats:
  - R-format
  - I-format
  - J-format

Instruction Formats

- I-format: used for instructions with immediates, **`lw`** and **`sw`** (since offset counts as an immediate), and branches (**`beq`** and **`bne`**) since branches are "relative" to the PC
  - (but not the shift instructions)
- J-format: used for **`j`** and **`jal`**
- R-format: used for all other instructions
- It will soon become clear why the instructions have been partitioned in this way

R-Format Instructions (1/5)

- Define “fields” of the following number of bits each: 6 + 5 + 5 + 5 + 5 + 6 = 32

6	5	5	5	5	6

- For simplicity, each field has a name:

`opcode`	`rs`	`rt`	`rd`	`shamt`	`funct`

- Important: On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer
  - Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63

R-Format Instructions (2/5)

- What do these field integer values tell us?
  - opcode: partially specifies what instruction it is
    - Note: This number is equal to 0 for all R-Format instructions
  - funct: combined with opcode, this number exactly specifies the instruction

- Question: Why aren't **`opcode`** and **`funct`** a single 12-bit field?
  – We'll answer this later

R-Format Instructions (3/5)

- More fields:
  - **`rs`** (Source Register): usually used to specify register containing first operand
  - **`rt`** (Target Register): usually used to specify register containing second operand (note that name is misleading)
  - **`rd`** (Destination Register): usually used to specify register which will receive result of computation

R-Format Instructions (4/5)

- Notes about register fields:
  - Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.
  - The word “usually” was used because there are exceptions that we’ll see later

R-Format Instructions (5/5)

- Final field:
  - shamt: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits (so it can represent the numbers 0-31)
  - This field is set to 0 in all but the shift instructions
- For a detailed description of field usage for each instruction, see green insert in COD
  - (We will provide a copy of the "green sheet" on all exams)

R-Format Example (1/2)

- MIPS Instruction:
  - `add $8,$9,$10`

- opcode = 0 (look up in table in book)
- funct = 32 (look up in table in book)
- rd = 8 (destination)
- rs = 9 (first operand)
- rt = 10 (second operand)
- shamt = 0 (not a shift)

R-Format Example (2/2)

opcode	rs	rt	rd	shamt	funct

- MIPS Instruction:
  - add $8,$9,$10
  - Decimal number per field representation:

0	9	10	8	0	32

  - Binary number per field representation:

000000	01001	01010	01000	00000	100000

hex

  - hex representation:
  - `0x012A 4020`
  - Called a Machine Language Instruction